栈(Stack)

上一篇我们说到了列表，它是一种最自然的数据组织方式，如果对数据的存储顺序要求不重要，那么列表就是一种非常适合的数据结构，但对于计算机其他的一些应用（比如后缀表达式），那么列表就显得有些无能为力， 所以，我们需要一种和列表功能相似但更复杂的数据结构。

栈，又叫堆栈，是和列表类似的一种数据结构，但是却更高效，因为栈内的元素只能通过列表的一端访问，称为栈顶，数据只能在栈顶添加或删除，遵循 先入后出(LIFO，last-in-first-out) 的原则，普遍运用于计算机的方方面面。

对栈的操作主要有两种，一是将一个元素压入栈，push方法，另一个就是将栈顶元素出栈，pop方法。

除此之外，栈还有其他的一些属性和方法：查看当前栈顶的元素值，我们使用 peek 方法，它仅仅返回栈顶元素值，并不删除它；clear 方法用于清空当前栈内的所有元素；top属性记录当前栈顶位置；length方法返回当前栈内元素总数等；接着我们定义栈的数据类型，并利用JS中的数组去实现它。

类型	描述
top(属性)	查看当前栈顶位置
pop(方法)	删除栈顶元素
push(方法)	将新元素入栈
peek(方法)	查看当前栈顶元素
length(方法)	查看当前栈元素总数
clear(方法)	清空栈内元素

栈数据类型定义

栈的实现

```
//定义栈

function Stack () {
    this.dataStore = [];    //初始化为空
    this.top = 0;           //记录栈顶位置
    this.pop = pop;         //出栈
```

```
    this.push = push;       //入栈
    this.peek = peek;       //查看栈顶元素
    this.length = length;   //查看栈内元素总数
    this.clear = clear;     //清空栈
}
```

我们利用 dataStore 来保存栈内元素，初始化为空数组，top 属性用于记录当前栈顶位置，初始化的时候为0，
表示栈顶对应数组的起始位置是0，如果有元素入栈，则该属性会随之反生变化。
首先我们先来实现第一个入栈方法。
push：向栈内压入一个新的元素

```
//该方法将一个新元素入栈，放到数组中 top 所对应的位置上，并将 top 的值加 1，让其指向数组的下一个空位置

function push( element ){
    this.dataStore[this.top++] = element;
}
```

能入栈，就得可以出栈，接着我们来看出栈方法：
pop：取出栈顶元素

```
//该方法与入栈相反，返回栈顶元素，并将 top 的值减 1

function pop(){
    return this.dataStore[--this.top];
}
```

如何查看栈顶元素呢，peek方法！
peek：查看栈顶元素

```
//该方法返回的是栈顶元素，即 top - 1 个位置元素
```

```
function peek(){
    if( this.top > 0 ) return this.dataStore[this.top-1];
    else return 'Empty';
}
```

这里我做了个判断，如果一个空栈调用了 peek 方法，因为栈内没有任何元素，所以我这里返回了一个 'Empty';
现在，我们已经有了基本的入栈、出栈、查看栈顶元素的方法，我们不妨试一试。

```
//初始化一个栈
var stack = new Stack();
console.log( stack.peek() );    // Empty

//入栈
stack.push('Apple');
stack.push('Banana');
stack.push('Pear');

//查看当前栈顶元素
console.log( stack.peek() );    // Pear
console.log( stack.pop() );     // Pear
```

如果我放入了一些水果，吃掉了一个，我现在想知道我还剩多少个水果怎么办？length 方法可以实现
length：返回栈内元素总数

```
//该方法通过返回 top 属性的值来返回栈内总的元素个数

function length(){
    return this.top;
}
```

我们把代码恢复到出栈前的状态，也就是里面已经放了三个水果，接着我们来看看

```
console.log( stack.length() );      // 3

//出栈
stack.pop();

console.log( stack.length() );      // 2
```

好了，我们还剩最后一个clear方法，我们来实现一下

clear：清空栈

```
//该方法实现很简单，我们将 top 值置为 0 ，  dataStore 数值清空即可

function clear(){
    delete this.dataStore;
    this.dataStore = [];
    this.top = 0;
}
```

我们将上面的栈清空试试

```
stack.clear();

console.log( stack.length() );      // 0
console.log( stack.peek() );        // Empty
```

至此，我们已经可以用JS实现一个栈，但是你仍可能处于不知道如何正确使用的状态，接下来，我们举两个例子，一起看看栈的使用。

案例1：实现数制间的相互转换

我们可以利用栈将一个数字从一种数制转换成另一种数制。例如将数字 n 转换成以 b 为基数的数字，可以采用如下算法（该算法只针对基数为 2-9 的情况）：

1. 最高位为 n % b ， 直接压入栈;
2. 使用 n / b 来代替 n ;

3. 重复上面的步骤，知道 n 为 0 ，并且没有余数；
4. 以此将栈内元素弹出，直到栈空，并依次将这些元素排列，就得到了转换后的形式

代码如下：

```
//进制转换（2-9）

function mulBase ( num , base ) {
    var s = new Stack();
    do{
        s.push( num % base );
        num = Math.floor( num /= base );
    }while ( num > 0 );

    var converted = '';
    while (s.length() > 0){
        converted += s.pop();
    }
    return converted;
}

console.log( mulBase( 125 , 2 ) );      // 1111101
console.log( mulBase( 125 , 8 ) );      // 175
```

案列2：判断一个字符串是不是回文

回文是指一个字符串，从前往后写和从后往前写结果都是一样的，比如单词 'level' ， 'racecar'，就是回文，数字 1001 也是回文。

我们采用栈，可以很轻松判断一个字符串是否是回文，实现算法很简单，相信你们都猜到了。我们把字符串从左到右依次压入栈，这样，栈中保存了该字符串反转后的字符，我们再依次出栈，通过比较出栈后的字符串是否与原字符串是否相等，就可判断该字符串是否是回文。

具体代码实现如下：

```
//回文判断

function isPalindrome ( word ) {
    var s = new Stack();
    for( var i = 0 ; i < word.length ; i ++ ){
```

```
        s.push( word[i] );
    }
    var rword = '';
    while( s.length() > 0 ){
        rword += s.pop();
    }

    if( word == rword ){
        return true;
    }else{
        return false;
    }
}

console.log( isPalindrome('level') )    // true
console.log( isPalindrome('1001') )     // true
console.log( isPalindrome('word') )     // false
```

本文主要讲的是栈的运用，所以采用上述方式判断字符串是否是回文，实际上，你完全可以采用以下方式更方便的判断一个字符串是否是回文：

```
function isPalindrome ( word ){
    return String(word).split('').reverse().join('') == word ? true : false;
}
```